

# Utilisation de COM et Interopérabilité

---

## Sommaire

Utilisation de COM et Interopérabilité .....	1
1 Introduction.....	2
2 Les composants COM et le code non-managé.....	3
2.1 Utilisation des objets COM.....	3
2.2 Gestion des exceptions COM .....	5
2.3 Utilisation de code non-managé sans les composants COM .....	5
2.3.1 Les bases de P/Invoke .....	5
2.3.2 Conversions de données : Le Marshaling.....	8
2.3.3 Compléments .....	11
3 Interopérabilité du code .NET .....	12
4 Conclusion .....	14

## 1 Introduction

Dans le chapitre 8, nous avons vu comment charger du code écrit pour fonctionner avec le .NET Framework dans notre application. Cela nous permettait de lancer d'autres applications .NET à partir d'une application .NET.

Seulement, il est possible que nous ayons à développer des applications en utilisant un autre système que le .NET Framework.



Pour cela, nous utiliserons un type d'assembly particulier : Les objets COM. Les objets COM sont des programmes écrits en code dit non-managé ; c'est-à-dire qu'ils n'utilisent pas le système de gestion des ressources proposé par le CLR du Framework .NET.

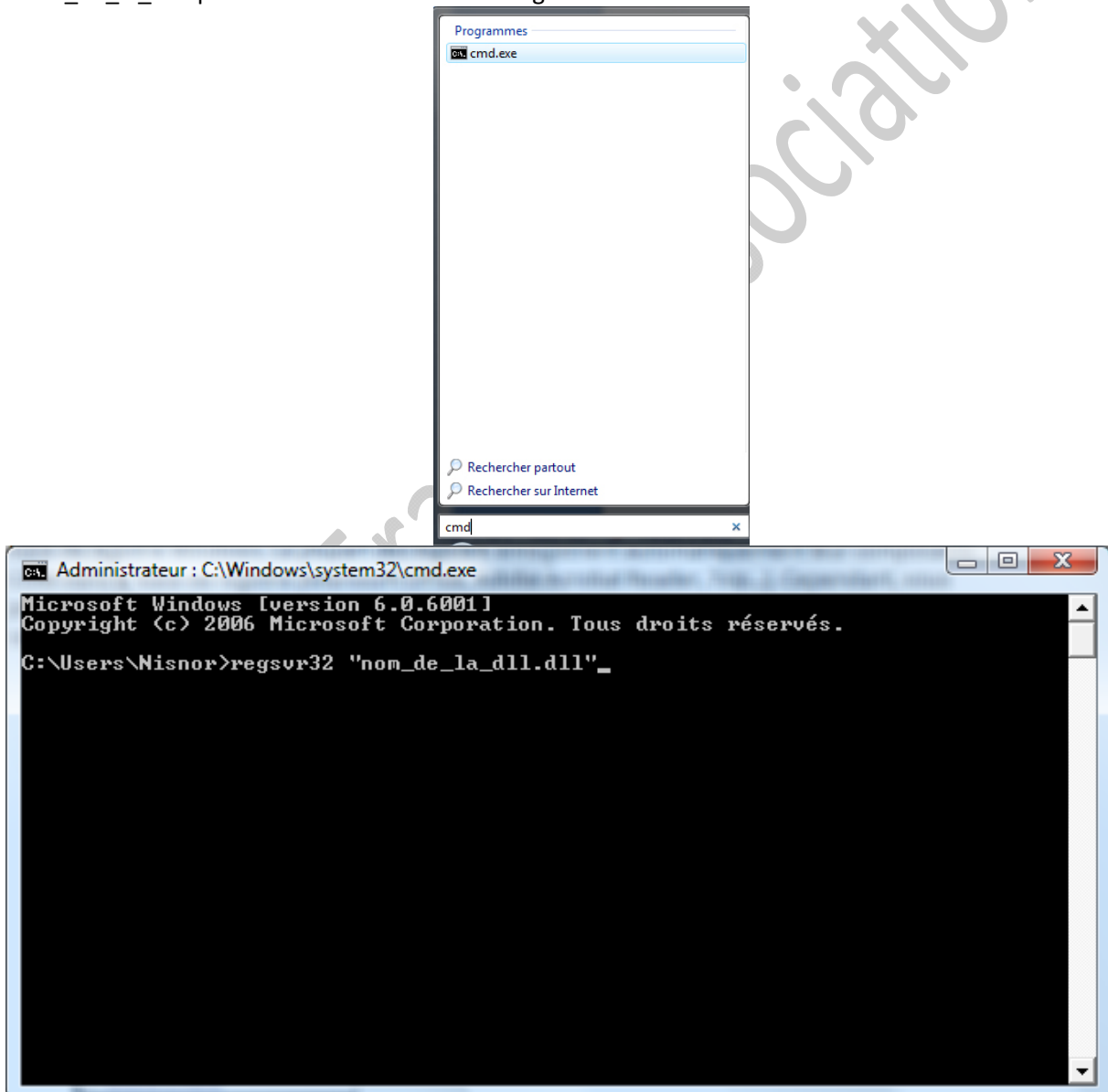
Dans cette partie, nous verrons comment importer des objets COM dans notre projet et comment se servir d'applications écrites en code managé dans des applications en code non-managé.

## 2 Les composants COM et le code non-managé

Les composants COM (Components Object Model) permettent aux applications prévues pour fonctionner avec le .NET Framework d'interagir avec des applications dites non-managées. Ces applications non-managées sont toutes les applications qui n'utilisent pas la CLR du .NET. Leur type de gestion des ressources est propre au développeur qui les a créées.

### 2.1 Utilisation des objets COM

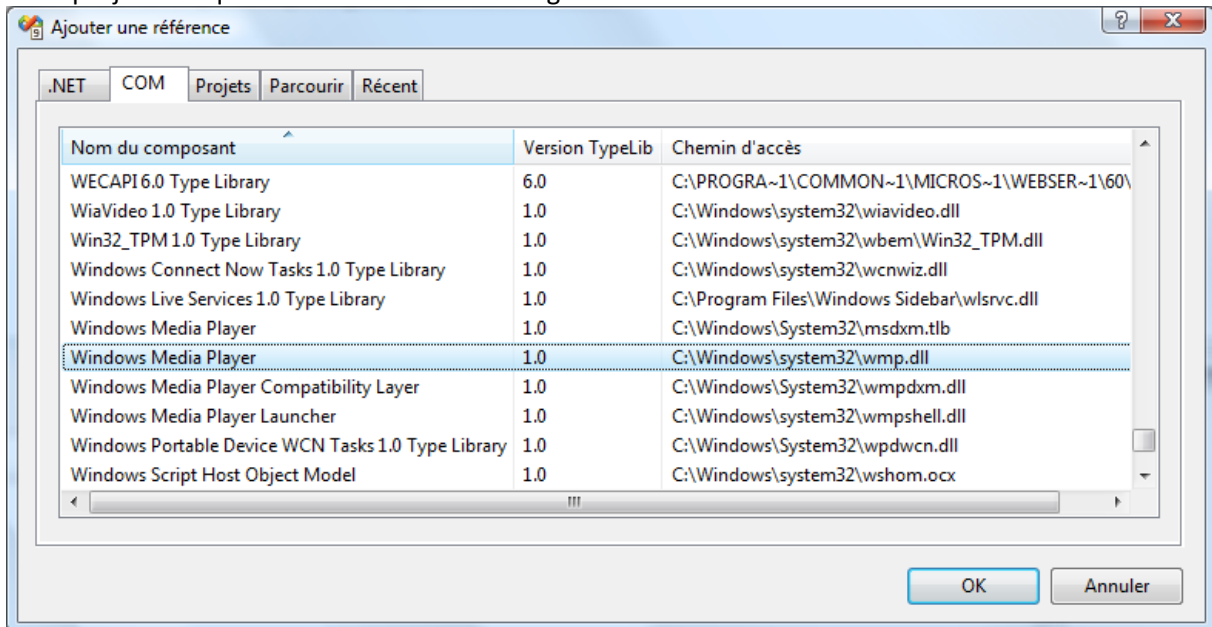
Pour pouvoir utiliser un composant COM dans vos applications, il doit être enregistré dans la base de registre Windows. La plupart des logiciels enregistrent automatiquement leur composants COM dans la base de registre (Microsoft Office, Adobe Acrobat Reader, 7zip...). Cependant, vous pourriez vouloir enregistrer vos propres créations. Pour cela, vous pouvez ouvrir une invite de commande de Windows et saisir la commande "regsvr32 "nom\_de\_la\_dll.dll" " en remplaçant <nom\_de\_la\_dll> par le nom de votre DLL à enregistrer :



**Note :** Si vous souhaitez ne plus utiliser votre composant COM, vous pouvez également le dés-enregistrer en utilisant le commutateur "/u" : regsvr32 /u "nom\_de\_la\_dll.dll". Vous pouvez obtenir la liste des commutateurs disponible juste en saisissant "regsvr32". Un message d'erreur s'affichera avec la liste des commutateurs disponibles.

Pour utiliser votre composant COM dans vos codes, vous pouvez soit passer par du code en utilisant les outils de System.Runtime.InteropServices (ce qui n'est pas forcément recommandé car c'est une source importante d'erreurs), soit passer par l'outil d'importation Visual Studio (Seule cette méthode sera expliquée ici).

Pour importer votre composant COM, vous faites comme si vous importiez une assembly dans votre projet sauf que vous sélectionnez l'onglet "COM" :



Vous n'avez plus qu'à sélectionner le composant que vous souhaitez importer et valider.

Vous pouvez également utiliser l'outil en ligne de commande TlbImp.exe dans l'invite de commande de Visual Studio. Celui-ci va transformer la dll de votre objet COM en assembly .NET, vous n'aurez plus qu'à ajouter une référence vers l'assembly créée.

Pour transformer votre dll voici la commande :

```
tlbimp votre_dll.dll
```

Ou

```
tlbimp votre_dll.dll /out:nouveau_nom.dll
```

La première va créer une assembly possédant le même nom que votre dll, la seconde vous permet de choisir le nom de l'assembly.

Pour vous donner un exemple, nous allons créer une application qui va lancer la lecture d'une piste audio en utilisant Windows Media Player. Pour cela, importez le composant COM nommé "Windows Media Player" et qui se situe à "C:\Windows\system32\wmp.dll". Ensuite, vous saisissez le code suivant :

```
'VB
Imports WMPLib

Module partiel
    Dim player As WMPLib.WindowsMediaPlayer
    Sub Main()
        player = New WindowsMediaPlayer()
        player.openPlayer("ftp://ftp2.mp3trazaac.com/mptrazaac/The
maze.mp3")
    End Sub
End Module
```

```
//C#
using WMPLib;

public static WindowsMediaPlayer player;

static void Main(string[] args)
{
    player = new WindowsMediaPlayer();
    player.openPlayer("ftp://ftp2.mp3trazaac.com/mptrazaac/The
maze.mp3");
}
```

Ce code se contente de créer une nouvelle instance du lecteur Windows Media Player et de lancer la lecture d'une piste MP3.

Malgré les efforts effectués pour que l'interopérabilité avec COM soit la plus performante et transparente possible, il existe des limites qu'il est nécessaire de connaître.

Tout d'abord les membres statiques ne sont pas supportés, dû aux différences de type évidentes entre le .NET et les objets COM.

Ensuite, vous ne pouvez pas utiliser de constructeur avec des paramètres, tous vos constructeurs doivent être ceux par défauts.

L'héritage est également limité, si dans une classe héritée certains membres masquent les membres de bases, vous serez incapable d'appeler les membres de bases.

Enfin, l'interopérabilité entre .NET et COM utilisant le registre Windows, elle ne fonctionne que sous Windows.

## 2.2 Gestion des exceptions COM

Lorsque vos applications sont créées pour fonctionner avec le Framework .NET, le système d'exception utilise un système conforme avec le CLS (Common Language Specifications). Mais lorsque vous exécutez une application COM, rien ne garanti que celle-ci soit également conforme avec le CLS.

Depuis la version 2.0 du Framework, le CLR instancie indifféremment la classe Exception, que ça soit pour une exception conforme avec CLS ou non. Ainsi, vous pouvez gérer vos erreurs de la même façon que pour du code managé ; un simple bloc try-catch suffira.

## 2.3 Utilisation de code non-managé sans les composants COM

Nous avons vu dans la partie précédente comment utiliser des objets COM dans vos assemblies. Il peut arriver que le composant COM englobant un code non managé n'existe pas, soit parce qu'il est déprécié soit parce que le code n'a pas été porté. Nous allons donc devoir utiliser d'autres outils du Framework pour importer et utiliser du code non managé.

### 2.3.1 Les bases de P/Invoke

L'outil principal pour manipuler du code managé est Platform Invoke (ou P/Invoke), il est situé dans l'espace de nom *System.Runtime.InteropServices*.

Pour utiliser P/Invoke, nous allons utiliser conjointement un attribut et le mot clef extern ou Shared en VB.NET:

```
'VB
<DllImport("ma.dll")> _
Private Shared Function MaMethode() As Int32

//C#
[DllImport("ma.dll")]
private static extern Int32 MaMethode();
```

L'attribut va se charger d'importer un membre de la Dll, vous devez le placer devant votre méthode ou un paramètre.

Le mot clef extern/Shared lui permet de définir que nous allons utiliser une méthode externe à l'assembly et que celle-ci se situe dans la dll importée. Vous devez impérativement respecter la signature de la méthode, vous devez donc connaître la signature de la méthode dans la dll importé !

Voici un exemple d'utilisation. Nous allons nous servir de l'API Win32 qui permet à la manière des Windows Form d'afficher des fenêtres ou dessiner des formes mais avec du code non managé .NET.

Dans ce code, nous allons simplement afficher une MessageBox :

```
'VB
Imports System.Runtime.InteropServices
Imports System.Text

Module partie13
  Class Win32Invoke
    Private Const buffer As Int32 = 256

    <DllImport("user32.dll")> _
    Private Shared Function GetForegroundWindow() As IntPtr
    End Function

    <DllImport("user32.dll")> _
    Private Shared Function MessageBox(ByVal hWnd As IntPtr, ByVal
texte As StringBuilder, ByVal titre As StringBuilder, ByVal constantes As
Int32) As Int32
    End Function

    Public Shared Sub Afficher()
      Dim texte As StringBuilder = New StringBuilder(buffer)
      texte.Append("Bienvenue dans ma Message Box")

      Dim titre As StringBuilder = New StringBuilder(buffer)
      titre.Append("Message Box")

      Dim fenetre As IntPtr = GetForegroundWindow()

      MessageBox(fenetre, texte, titre, 4)
    End Sub

  End Class

  Sub Main()
    Win32Invoke.Afficher()
  End Sub
End Module
```



```
//C#
using System.Runtime.InteropServices;
static class Win32Invoke
{
    private const Int32 buffer = 256;

    [DllImport("user32.dll")]
    private static extern IntPtr GetForegroundWindow();

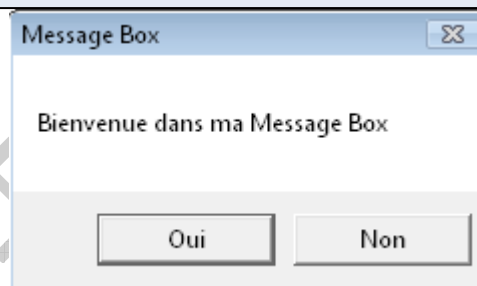
    [DllImport("user32.dll")]
    private static extern Int32 MessageBox(IntPtr hWnd, StringBuilder
texte, StringBuilder titre, Int32 constantes);

    public static void Afficher()
    {
        StringBuilder texte = new StringBuilder(buffer);
        texte.Append("Bienvenue dans ma Message Box");

        StringBuilder titre = new StringBuilder(buffer);
        titre.Append("Message Box");

        IntPtr fenetre = GetForegroundWindow();

        MessageBox(fenetre, texte, titre, 4);
    }
}
static void Main(string[] args)
{
    Win32Invoke.Afficher();
}
```



Nous avons créé une classe statique Win32Invoke, celle-ci est chargée de factoriser le code permettant d'afficher notre MessageBox.

A l'intérieur nous allons importer les deux méthodes permettant d'afficher la MessageBox, la première, GetForegroundWindow retourne un pointeur vers la fenêtre principale, ou, comme dans notre cas s'il n'y a pas de fenêtre principale, cela pointe vers une valeur à null.

Notre deuxième méthode permet d'afficher la MessageBox, nous avons respecté sa signature conformément à une [documentation trouvée en ligne](#).

Ensuite dans une méthode Afficher nous appelons nos deux méthodes.

Nous utilisons des StringBuilder plutôt que des String car ils fonctionnent de façon dynamique, et sont donc plus performants quand on utilise du code non-managé.

**Note :** Vous pourrez trouver les prototypes des méthodes de ces API en allant sur le [MSDN](#) dans la section Win32 and COM development.

Nous avons dans notre exemple encapsulé le code importé dans une classe et une méthode afin de simplifier la réutilisation du code non managé. C'est une bonne pratique car elle va vous permettre d'importer du code non managé sans perturber les autres développeurs qui pourront utiliser seulement du code managé et ne devront pas apprendre toutes les ficelles de P/Invoke.

Factoriser le code dans une classe va aussi vous permettre d'utiliser les subtilités du C# afin de limiter les erreurs de type induit par des langages trop permissifs. En utilisant par exemple les generics vous vous assurez de respecter la signature d'une méthode.

### 2.3.2 Conversions de données : Le Marshaling

Nous avons vu précédemment comment utiliser du code non managé, mais nous avons un petit peu rusé pour cela. En effet si vous comparez la signature de la méthode MessageBox et celle que nous avons défini, elle est un petit peu différente. Nous utilisons par exemple des StringBuilder à la place de LPCTSTR et Int32 à la place de UINT.

Le Marshaling va nous permettre dans l'essentiel de convertir les types de données entre code managé et non managé.

Il est très important que vous utilisiez le Marshaling car passer des types managés aux types non managés peut induire de graves erreurs.

Afin de convertir des types de données simple nous allons utiliser l'attribut MarshalAs. Celui-ci peut s'appliquer à un type de retour, à une variable ou à un paramètre de fonction.

Nous avons pour l'exemple modifié notre précédent exemple en utilisant le Marshaling :



```
'VB
Class Win32Invoke
    Private Const buffer As Int32 = 256

    <MarshalAs(UnmanagedType.LPStr)> _
    Private texte As StringBuilder

    <MarshalAs(UnmanagedType.LPStr)> _
    Private titre As StringBuilder

    <DllImport("user32.dll")> _
    Private Shared Function GetForegroundWindow() As IntPtr
    End Function

    <DllImport("user32.dll")> _
    Private Shared Function MessageBox(ByVal hWnd As IntPtr,
    <MarshalAs(UnmanagedType.LPStr)> ByVal texte As StringBuilder,
    <MarshalAs(UnmanagedType.LPStr)> ByVal titre As StringBuilder, ByVal
    constantes As Int32) As Int32
    End Function

    Public Shared Sub Afficher()
        Dim texte As StringBuilder = New StringBuilder(buffer)
        texte.Append("Bienvenue dans ma Message Box")

        Dim titre As StringBuilder = New StringBuilder(buffer)
        titre.Append("Message Box")

        Dim fenetre As IntPtr = GetForegroundWindow()

        MessageBox(fenetre, texte, titre, 4)
    End Sub
End Class
```

Dotnet-France

```
//C#
static class Win32Invoke
{
    private const Int32 buffer = 256;
    [MarshalAs(UnmanagedType.LPStr)]
    private static StringBuilder texte;

    [MarshalAs(UnmanagedType.LPStr)]
    private static StringBuilder titre;

    [DllImport("user32.dll")]
    private static extern IntPtr GetForegroundWindow();

    [DllImport("user32.dll")]
    private static extern Int32 MessageBox(IntPtr hWnd,
    [MarshalAs(UnmanagedType.LPStr)] StringBuilder texte,
    [MarshalAs(UnmanagedType.LPStr)] StringBuilder titre, Int32 constantes);

    public static void Afficher()
    {
        texte = new StringBuilder(buffer);
        texte.Append("Bienvenue dans ma Message Box");

        titre = new StringBuilder(buffer);
        titre.Append("Message Box");

        IntPtr fenetre = GetForegroundWindow();

        MessageBox(fenetre, texte, titre, 4);
    }
}
```

Nous avons donc utilisé l'attribut `MarshalAs` devant nos attributs `texte` et `titre`, ainsi que devant le deuxième et troisième paramètre de `MessageBox`. Si nous utilisons un type incompatible, le résultat sera soit imprévisible, soit il engendrera une exception.

Grâce à l'intellisense de Visual Studio, vous pouvez connaître l'ensemble des types disponibles en tapant `UnmanagedType` suivi d'un point. Vous pouvez également vous rendre sur [MSDN](#).

Enfin, nous n'avons présenté ici que le Marshaling de type de données simple, vous trouverez un bon nombre d'exemples et de tutoriaux sur le Marshaling sur [MSDN](#).

### 2.3.3 Compléments

Nous avons présenté une introduction à la gestion du code non managé en .NET, vous trouverez de nombreuses ressources sur internet pour aller plus loin.

Sachez néanmoins que vous pourrez utiliser la plupart des outils du .NET pour gérer votre code managé, Génériques, Collections, mais aussi delegates (pour les callback) ou même les exceptions.

Enfin, veillez à faire attention lorsque vous utilisez du code non managé. Si les performances sont tout à fait honorables, on ne peut pas en dire autant de la sécurité du code et des problèmes de typage (surtout dans les langages très permissifs comme le C ou le C++). Testez donc bien vos codes, analysez-les, et prenez connaissance d'un maximum d'information grâce aux documentations des codes non managés.

Cela étant dit, si vous développez avec rigueur, il n'y a pas de raison de s'en priver. En effet, vous allez pouvoir utiliser dès à présent n'importe quelle DLL non managé avec votre code managé et ainsi rendre compatible des bibliothèques sympathiques telles que [SDL](#), ou bien encore [Havok](#).

Dotnet-France Association

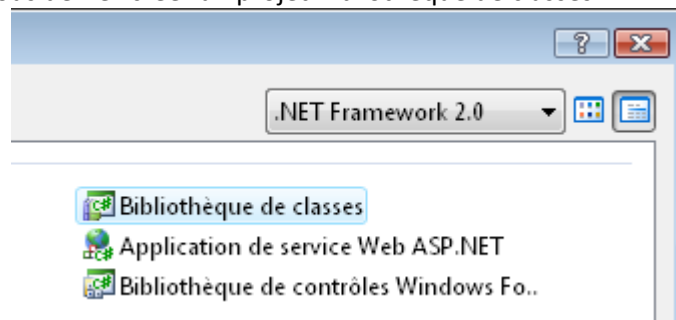
### 3 Interopérabilité du code .NET

Dans la première partie nous avons vu comment importer un objet COM dans votre code .NET et ainsi permettre d'utiliser le code d'applications ou de librairie qui ne sont pas du .NET dans vos applications .NET. Nous allons maintenant voir l'effet inverse, c'est-à-dire transformer vos assemblies de telle manière à ce que vous puissiez les utiliser dans vos objets COM.

Pour rendre vos assemblies interopérable avec des objets COM, il existe un manager capable de faire interface entre une assembly .NET et un objet COM. Cette interface se nomme COM Callable Wrapper (CCW). Elle va se charger de distribuer votre assembly sous forme [marshalisée](#) aux différents objets COM.

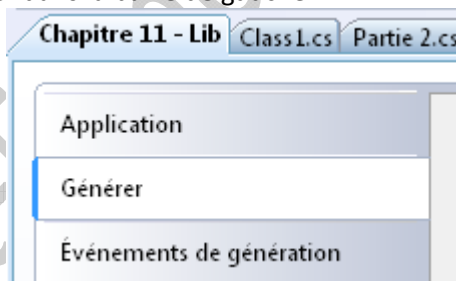
Pour rendre votre assembly utilisable par des objets COM, c'est du côté de Visual Studio que nous allons travailler.

Tout d'abord vous devrez créer un projet Bibliothèque de classes :



Ensuite créez les classes que vous souhaitez rendre compatible, par exemple une classe personnage contenant trois attributs et trois propriétés.



Enfin, faites un clic droit sur votre projet, puis propriété. Dans le panneau de configuration ainsi ouvert, cliquez sur Générer dans la barre de gauche :



Enfin en bas de la page, cliquez sur Inscrire pour COM Interop :

Inscrire pour COM Interop

Vous pouvez maintenant générer votre projet. Une dll est maintenant créée, vous allez pouvoir la faire interagir avec vos composants COM.

Nom	Date de modificati...	Type
 Chapitre 11 - Lib.dll	14/08/2008 10:17	Extension de l'app...
 Chapitre 11 - Lib.pdb	14/08/2008 10:17	Program Debug D...

Quelques Attributs vont rendre visibles ou non certaines parties du code. Pour cela, nous allons utiliser les espaces de nom et l'attribut suivant :

```
'VB
' Espaces de nom :
Imports System.Runtime.CompilerServices
Imports System.Runtime.InteropServices

' Attribut :
<ComVisible(booléen)>

//C#
//Espaces de nom :
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

//Attribut :
[ComVisible(booléen)]
```

L'attribut ComVisible peut être placé devant vos classes et vos membres de classes, si vous voulez masquer une partie des membres, vous devrez rendre votre classe non visible puis faire une liste blanche des membres visibles.

Sachez enfin que votre code doit respecter certaines conventions afin que la compatibilité soit bonne. Votre classe devra comporter au moins un constructeur sans paramètre et seules les classes et membres publics peuvent être visibles.

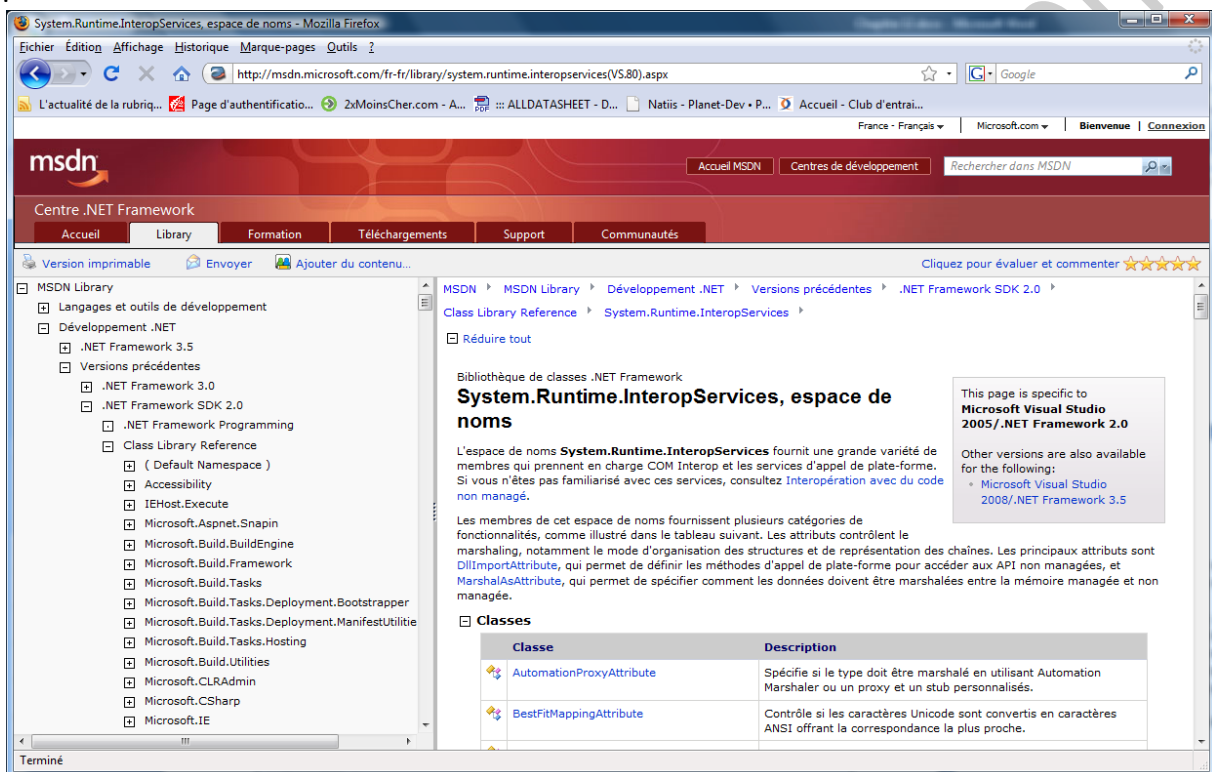
## 4 Conclusion

Au cours de ce chapitre, vous avez pu constater la réelle simplicité d'utilisation des codes non-managés dans vos applications .NET

A la fin de ce chapitre, vous devriez pouvoir :

- Importer un objet COM, comment les utiliser et comment utiliser des applications .NET dans des codes non-managés.
- Gérer les exceptions des composants COM.
- Utiliser des codes non-managés sans passer par les composants COM.

Dans tous les cas, le [MSDN](http://msdn.microsoft.com) peut vous apporter un soutien de développement non négligeable



The screenshot shows the MSDN website in a Mozilla Firefox browser window. The page title is "System.Runtime.InteropServices, espace de noms". The breadcrumb trail is "MSDN > MSDN Library > Développement .NET > Versions précédentes > .NET Framework SDK 2.0 > Class Library Reference > System.Runtime.InteropServices".

The main content area displays the title "Bibliothèque de classes .NET Framework" followed by "System.Runtime.InteropServices, espace de noms". Below this, there is a paragraph of introductory text and a section titled "Classes" containing a table with two columns: "Classe" and "Description".

Classe	Description
AutomationProxyAttribute	Spécifie si le type doit être marshalé en utilisant Automation Marshaler ou un proxy et un stub personnalisés.
BestFitMappingAttribute	Contrôle si les caractères Unicode sont convertis en caractères ANSI offrant la correspondance la plus proche.

On the right side of the page, there is a note: "This page is specific to Microsoft Visual Studio 2005/.NET Framework 2.0. Other versions are also available for the following: Microsoft Visual Studio 2008/.NET Framework 3.5".